# Fast Evaluation of CTEQ Parton Distributions in Monte Carlos

Zack Sullivan

*Theoretical Physics Department, Fermi National Accelerator Laboratory, Batavia, IL 60510-0500, USA*

**Abstract**

A few changes to the routines that calculate CTEQ parton distribution functions allow modern compilers to optimize the evaluations, while having no quantitative effect on the results. Overall computation time is reduced by a factor of 4–5 in matrix-element calculations, and by 1.3–2.5 in showering Monte Carlo event generators. Similar improvements in performance may be expected in any calculations relying heavily on interpolation or multiple calls to functions.

## 1 Introduction

A significant amount of time and computing resources are spent on calculating events at hadron colliders. Whether a theoretical calculation of matrix elements, or an experimental simulation of events with detector effects, one common element is the evaluation of parton distribution functions (PDFs). These functions return the probability of finding a parton (quark or gluon) inside of a proton, based on two parameters: the fraction of momentum carried by the parton $x$, and the square of the energy scale of the process $Q^2$. Because the input parameters can span several orders of magnitude, the values of these functions are stored in two-dimensional tables for a finite number of input points. An approximate result for an arbitrary input of $x$ and $Q^2$ is derived by interpolating between the values obtained from the nearest table entries.

In profiling `ZTOP` [1,2], a `FORTRAN` code written to simulate next-to-leading-order jet distributions in single-top-quark production, it has become apparent that much of the execution time of real production code is spent acquiring PDFs. Upon close examination of the CTEQ4 and CTEQ5 PDF codes [3], a handful of trivial optimizations arise that can cut this time in half. Based on

Table 1
Fraction of time spent evaluating PDF functions using default CTEQ computer codes for three programs: the next-to-leading-order jet calculation `ZTOP`, and two showering event generators, `HERWIG` and `PYTHIA`.

| Program | CTEQ4/5 | CTEQ6 |
|---------|---------|-------|
| ZTOP    | 90%     | 60%   |
| HERWIG  | 70%     | 33%   |
| PYTHIA  | 35%     | 16%   |

this success, I examine further algorithmic improvements in the typical interface functions that reduce the execution time by another factor of two or more for all CTEQ PDFs (including CTEQ6 [4]). I provide specific recommendations that are simple to implement, but which can have large consequences for efficiency.

A `gprof` profile of `ZTOP` [1,2] indicates that up to 90% of the execution time is spent in acquiring PDFs. Execution times in other programs appear to be dominated by the same routines. In Table 1 I show the typical fraction of time spent evaluating PDFs for `ZTOP`, and the two most common showering event generators, `HERWIG` 6.1 [5], and `PYTHIA` 6.2 [6]. The results in Table 1 were generated using the GNU `g77` 3.1 compiler for linux on a 1.4 GHz Pentium 4 processor with the flags `-g -pg -O3 -march=pentium4 -msse2`, and were verified by commenting out the routines. The results vary by less than 3% when changing compilers or compiler flags. Times for CTEQ4 and CTEQ5 differ from CTEQ6, because the latter uses a different interpolation algorithm. Retrieving PDFs is always the most time-intensive operation in these calculations. Therefore, it behooves us investigate what options are available to speed up the PDF routines.

Having identified the PDFs as the main bottleneck in the calculation of cross sections and events at hadron colliders, I will examine several successive levels of optimization in Sec. 2. The trade off will be that each level involves replacing a larger fraction of the base code. In Sec. 3 I evaluate the effectiveness of each change using a benchmark program, and three real production codes: `ZTOP`, `HERWIG`, and `PYTHIA`. I conclude with some observations and recommendations for future improvements.

## 2   Levels of optimization

It is important to recognize that code which evaluates PDFs is often embedded in complex ways inside an application. Hence, the replacement of a given

routine could be considered invasive. I consider classes of optimization that each replace larger portions of code. In practice the first two changes I suggest are easy to accommodate. However, the final one replaces a routine from a commonly used library, and hence care must be taken to ensure that no hidden dependencies arise. In all cases the routines have been verified to work with all programs mentioned in this paper.

## 2.1 Modifying `POLINT`

Looking more carefully at a profile of `ZTOP` using CTEQ5 PDFs indicates that more than 75% of the time is spent inside the subroutine `POLINT`. `POLINT` is a routine designed to perform a polynomial fit of degree $n-1$ to a data set of $n$ points based on *Neville's algorithm* [7]. This subroutine is used by the CTEQ Collaboration [3,4] to interpolate smoothly between the values of $x$ and $Q^2$ that are read in from a table of best-fit values.

One approach to increasing speed would be to replace `POLINT` outright with alternate interpolations, or functional fits to the PDFs. While these are reasonable choices, it is important to ensure that any results are numerically identical to results obtained previously. Therefore, I begin by making trivial modifications of `POLINT` itself. Two useful changes [8] are:

(1) Remove the line: `IF(DEN.EQ.0.)PAUSE`.
(2) Write different versions of `POLINT` for 3- and 4-point interpolation, and call them directly. E.g., replace `POLINT(XA,YA,N,X,Y,DY)` with `POLINT3(XA,YA,3,X,Y,DY)` in `PARTONX4` or `PARTONX5`.

The first optimization is the most important as the line is never reached in the evaluation of the CTEQ PDFs, but it generally prevents the compiler from fully optimizing the loops [9]. Beyond being an unnecessary comparison, allowing a break point out of the loop forces the processor to flush the instruction pipeline, and can produce a missed branch comparison. It also prevents the compiler from using most types of parallel instructions. The second optimization mostly helps compilers to optimize the loops by defining the number of iterations at compile time, rather than dynamically.

This optimization is more often effective with CTEQ4/5 than with CTEQ6, since `POLINT` is only used at the edges of the $x$ and $Q^2$ tables in CTEQ6. However, using these optimizations with CTEQ6 will never be slower, and can be much faster for some calculations. We will see in Sec. 3 that using `POLINT3` and `POLINT4` has other benefits as well.

A further optimization for CTEQ4/5 comes from completely recoding *Neville's algorithm* for the special case of 3 points, and removing the return of the error

estimation. As a general routine, `POLINT` evaluates several expressions that are never used if there are only 3 points. Furthermore, the CTEQ code does not use the error estimate provided in the general case. Therefore, all unnecessary calculations and assignments are removed. This results in a reduction of the number of machine instructions, data reads, and data writes by a factor of 3. Net effects on the overall speed of execution are described in Sec. 3. The "fast" version of `POLINT3` is listed in the Appendix.

## 2.2  Modifying `PARTONXN`

Most programs that use parton distribution functions access them through interface routines, such as `STRUCTM` and `PFTOPDG` [10]. The key feature of these routines is that they ask for the density of all partons at once ($u_v$, $u_s$, $d_v$, $d_s$, $s$, $c$, $b$, $g$). Typically this is done by looping over the routines that access the PDFs, where the values of $x$ and $Q^2$ are fixed, but only the flavor of the parton changes. This immediately suggests an algorithmic improvement that should be applicable to all types of parton distributions: save the values of $x$ and $Q^2$, and the results of any functions applied to them, and bypass those functions unless $x$ or $Q^2$ change.

This algorithmic improvement in the CTEQ PDFs involves minor edits to the routines `PARTONXN`, where `N` is the number of the CTEQ set. The changes consist of adding a few `SAVE` statements, and a test for whether $x$ or $Q^2$ has changed. For CTEQ4/5 add

```
DOUBLE PRECISION XLAST, QLAST, QG
INTEGER JX,JQ
DATA XLAST, QLAST / -1D0, -1D0 /
DATA JX, JQ / 0, 0 /
SAVE XLAST,QLAST,QG,JX,JQ

IF ((X.EQ.XLAST).AND.(Q.EQ.QLAST))
&   GOTO 99
XLAST=X
QLAST=Q
```

after the declaration statements, and add a statement label 99 to the first line that involves the parton flavor:

```
99   IF (IPRTN .GE. 3) THEN
```

The calls to `POLINT` should also be changed to one of the versions of `POLINT3` mentioned in Sec. 2.1.

The CTEQ6 PDFs use a completely different interpolation through the table of $x$ and $Q^2$. Most of the calculations in `PARTONX6` are associated with this interpolation, and therefore there is a greater potential gain by adding

```
      DOUBLE PRECISION X, Q
      INTEGER JX, JQ
      DATA X, Q / -1D0, -1D0 /
      DATA JX, JQ / 0, 0 /
      SAVE X, Q, JX, JQ, JLX, JLQ
      SAVE SS, CONST1, CONST2, CONST3, CONST4
      SAVE CONST5, CONST6
      SAVE SY2, SY3, S23, TT, T12, T13, T23
      SAVE T24, T34, TY2, TY3
      SAVE TMP1, TMP2, TDET

      IF ((XX.EQ.X).AND.(QQ.EQ.Q)) GOTO 99
```

after the declaration statements, and adding the statement label 99 to the same place as in CTEQ4/5. The calls to `POLINT` should also be changed to `POLINT4`, as mentioned in the Sec. 2.1.

### 2.3  Modifying *STRUCTM* and *PFTOPDG*

All of the optimizations suggested so far consist of modifying the CTEQ routines. However, there can still be a significant overhead in calling the CTEQ routines. Since most programs access the PDFs by calling `STRUCTM` or `PFTOPDG`, a final improvement would be to completely replace these routines with versions specialized to the CTEQ PDFs. The idea is to incorporate `PARTONXN` directly into `STRUCTM`, and to remove any additional redundancies. These routines may be obtained from the author or from Ref. [11].

There are two options for optimizing `PFTOPDG`. The first option is to write another copy of the code in `STRUCTM` that returns the values in a different format. The second option, which is used by the `CERNLIB PDFLIB` routines, is to simply call `STRUCTM`. The first option is error-prone, and only provides a 2% improvement in speed. The second option is actually an interface bug in `CERNLIB PFTOPDG`, since it advertises that it will separately return the PDFs for all quarks and anti-quarks. This is fine for CTEQ4–6, but newer PDFs may not have the same value for $s$ and $\bar{s}$. Therefore, a call to `PFTOPDG` may quietly give incorrect results in the future. Care must be taken to ensure that a given set of PDFs are consistently accessed.

## 3 Optimization results

In order to assess the usefulness of these optimizations, I evaluate four programs. I consider a loop over PDFs, and three calculations of $t$-channel single-top-quark production: ZTOP, an analytic next-to-leading order calculation of jet distributions [1,2], HERWIG 6.1 [5], and PYTHIA 6.2 [6]. All calculations are compiled with the GNU FORTRAN compiler g77 versions 2.95 and 3.1, and the Intel compiler ifc 6.0. Previously [8], I have investigated the effects of the first optimization on HERWIG and PYTHIA while including fast detector simulation SHW [12]. The routines added by SHW contribute a few percent to the overall execution time. Hence, in order to more effectively isolate the effects of the PDFs and reduce dependency on unnecessary libraries, I do not use SHW here, or write out any data. All numerical results are from execution on a 1.4 GHz Pentium 4 machine. Limited tests performed on Pentium 3 machines are completely consistent with the results described below.

### 3.1 Benchmark for PDFs

The most naive test of potential speed gains comes from looping over parton distribution functions. I probe values of $x$ between $10^{-5}$ and 0.98, and fix the scale to be $Q^2 = (x * 1960 \text{ GeV})^2$. These choices avoid any possibility of anomalous gains due to fast memory access in the level 2 cache. However, this may underestimate the benefit of using POLINT4 in the CTEQ6 distributions for processes at the Large Hadron Collider at CERN, where larger values of $Q^2$ will be typical. In Ref. [8], I called the CTEQ routines directly. For this comparison, I access the routines using a simple version of STRUCTM. This is more representative of the typical use of the PDFs, and allows a direct comparison of all optimization levels.

In Table 2 I show the relative speed gain for each correction broken down by compiler and PDF set. The numbers are normalized to the results obtained using POLINT3/4. This choice is based on the observation in Ref. [8] that the execution times in typical codes are dependent on detailed choices of compiler flags. However, by using POLINT3/4, this dependence tends to disappear. Hence, by using the lowest level of optimization, not only do the programs operate up to factors of 2 faster, the speeds become more dependent on algorithms and less dependent on compiler details.

Table 2 demonstrates the speed enhancement due to saving common results between calls, as described in Sec. 2.2. This is a mild enhancement for CTEQ4/5, where the dominant subroutine is POLINT3, but is a significant improvement for CTEQ6. For CTEQ4/5, I also show the results of using a fully optimized

Table 2
Typical speed gains compared to `POLINT3/4` when looping over all partons, and $10^{-5} < x < 0.98$. Each column is separately normalized. The fastest time for each is also listed.

| Optimization | g77 3.1(2.95) | | ifc 6.0 | |
|---|---|---|---|---|
| | CTEQ4/5 | CTEQ6 | CTEQ4/5 | CTEQ6 |
| Default CTEQ | 1/(1.1–1.2) | 1.0 | 1/(1.5–2.7) | 1/1.03 |
| `POLINT3/4` | 1.0 | 1.0 | 1.0 | 1.0 |
| `POLINT3` (fast) | 1.5 | — | 1.2 | — |
| `SAVE` $x, Q^2$ | 1.26 | 2.6 | 1.12 | 2.4 |
| `SAVE` $x, Q^2$ (fast) | 2.3 | — | 1.9 | — |
| fastest `STRUCTM` | 3.1 | 3.1 | 4.6 | 2.7 |
| fastest times | 40 s | 50 s | 17 s | 35 s |

version of `POLINT3`, described in Sec. 2.1, and the total improvement when combined with saving the values. At this level, the typical gain over the base `POLINT3/4` is an additional factor of 2.

There are several possible ways to combine `POLINT` and `PARTONXN` into `STRUCTM`. The line labeled "fastest `STRUCTM`" lists the speed gain over `POLINT3/4` using all suggested improvements. The net enhancement over the default CTEQ distributions ranges from a factor of 3 to more than a factor of 12. All benchmarks are somewhat artificial, but this is a good indication of the upper range of improvements we might expect.

The final line of Table 2 shows the fastest times achieved for an arbitrary fixed number of loops. The first observation is that the `ifc` compiler tends to be a factor of 1–3 times faster than the `g77` compilers. (The difference between `g77` 2.95 and `g77` 3.1 tends to be less than 5–10%.) The second observation is that the fastest CTEQ4/5 is up to a factor of 2 faster than CTEQ6. This effect comes entirely from the difference in interpolation algorithms. We should therefore expect the fractions of time spent calling PDFs listed in Table 1 to become smaller, and CTEQ4/5 to use less time than CTEQ6.

### 3.2 Matrix element codes and *ZTOP*

Benchmarks can be misleading. Therefore, I consider the effects of the optimizations of Sec. 2 on a working production code of single-top-quark production [1,2] called `ZTOP`. The results listed in Table 3 show a remarkably similar gain to the benchmark scenario except at the fastest times. Again,

7

Table 3
Typical speed gains for the matrix-element code `ZTOP` relative to `POLINT3/4`. Each column is separately normalized. The fastest time for each is also listed.

| Optimization | g77 3.1(2.95) | | ifc 6.0 | |
| --- | --- | --- | --- | --- |
| | CTEQ4/5 | CTEQ6 | CTEQ4/5 | CTEQ6 |
| Default CTEQ | 1/1.2 | 1.0 | 1/(1.6–2.2) | 1.0 |
| `POLINT3/4` | 1.0 | 1.0 | 1.0 | 1.0 |
| `POLINT3` (fast) | 1.3 | — | 1.25 | — |
| `SAVE` $x, Q^2$ | 1.2 | 2.0 | 1.13 | 2.0 |
| `SAVE` $x, Q^2$ (fast) | 1.7 | — | 1.7 | — |
| fastest `STRUCTM` | 1.9 | 2.15 | 1.9, 2.7 | 2.15 |
| fastest times | 86 s | 98 s | 60, 42 s | 62 s |

the replacement of `POLINT` by `POLINT3` tends to remove the dependence on compiler flags. By using the replacement for `STRUCTM`, an additional factor of 2 is typical. The `ifc` compiler pushes this to a factor of 3 on a Pentium 4 processor by adding vectorization. The last line of Table 3 shows the typical result that the Intel compiler is a factor of 1.5–2 faster than `g77`.

Not all matrix element calculations use all PDFs. In the case of $t$-channel single-top-quark production, one leg in the matrix-element diagrams has only an incoming $b$ or $\bar{b}$ quark, or a gluon $g$. Additional time might be saved by eliminating any unnecessary calls to the PDFs. In practice this can be very difficult to achieve, e.g., `HERWIG` and `PYTHIA` use the `PDFLIB STRUCTM` [10] interface as an abstraction. In order to use individual PDFs, they would have to incorporate a new interface. In the case of `ZTOP`, the execution time can be reduced by a factor of 1.5 from the base `POLINT3/4`, but the improvement is less significant as additional optimizations are used. In general, if there is a clear way to eliminate extraneous calls to PDFs when coding matrix elements, it should be implemented.

### 3.3  `HERWIG` and `PYTHIA`

Theoretical calculations are typically at the matrix-element or jet level, but experimental and careful phenomenological studies generally resort to using showering Monte Carlo event generators. These codes are significantly more complex, and we should expect to see less gain in efficiency as additional time is spent in showering and detector simulation. In order to assess the impact on the two most common event generators, `HERWIG` 6.1 [5] and `PYTHIA` 6.2 [6], I use them to calculate $t$-channel single-top-quark production, including all

Table 4

Typical speed gains for PYTHIA relative to POLINT3/4. Each column is separately normalized. The fastest time for each is also listed.

| Optimization | g77 3.1(2.95) | | ifc 6.0 | |
| --- | --- | --- | --- | --- |
| | CTEQ4/5 | CTEQ6 | CTEQ4/5 | CTEQ6 |
| Default CTEQ | 1/1.03 | 1.0 | 1/(1.15–1.25) | 1.0 |
| POLINT3/4 | 1.0 | 1.0 | 1.0 | 1.0 |
| POLINT3 (fast) | 1.06 | — | 1.05 | — |
| SAVE $x, Q^2$ | 1.04 | 1.13 | 1.05 | 1.13 |
| SAVE $x, Q^2$ (fast) | 1.1 | — | 1.1 | — |
| fastest STRUCTM | 1.14 | 1.15 | 1.18 | 1.15 |
| fastest times | 55 s | 58 s | 42 s | 43 s |

showering effects.

While evaluating PDFs is indeed the most time-intensive operation in PYTHIA, Table 1 tells us that no optimization can provide more than about a factor of 1.5 improvement in speed. In Table 4 we see that the ifc compiler achieves almost the full factor of 1.5, while g77 can attain a factor of 1.2. Both results are significantly faster than the parameterizations of CTEQ PDFs built into PYTHIA. Since using table-based PDFs is also inherently more accurate, there appears to be no reason to continue producing parameterizations.

The HERWIG event generator spends almost as much time evaluating PDFs as the matrix element example considered above. This can be traced to HERWIG calling STRUCTM 1100–1800 times for each event requested. Table 5 shows the speed gains when using the each level of optimization as before. Remarkably, execution times are reduced by a factor of 1.8–2.5 for CTEQ4/5, and by 1.4–1.7 for CTEQ6. The actual times are somewhat anomalous, however. Unlike the benchmark, ZTOP, or PYTHIA, g77 3.1 appears to produce faster code than ifc 6.0, and CTEQ6 is faster than CTEQ4/5. This is fortuitous for people using the g77 compiler, but there is a catch. Using any compiler, and any PDF code (including the default CTEQ PDFs), the number of events produced by HERWIG 6.1 differs by up to 5% when different compiler flags are used. There appears to be a large sensitivity to round-off errors when using initial-state showering. This should be further investigated to determine whether this issue affects all HERWIG results, or is isolated to certain processes.

Table 5

Typical speed gains for `HERWIG` relative to `POLINT3/4`. Each column is separately normalized. The fastest time for each is also listed.

| Optimization | g77 3.1(2.95) | | ifc 6.0 | |
| --- | --- | --- | --- | --- |
| | CTEQ4/5 | CTEQ6 | CTEQ4/5 | CTEQ6 |
| Default CTEQ | 1/1.12 | 1.0 | 1/(1.2–1.7) | 1.0 |
| `POLINT3/4` | 1.0 | 1.0 | 1.0 | 1.0 |
| `POLINT3` (fast) | 1.25 | — | 1.1 | — |
| `SAVE` $x, Q^2$ | 1.14 | 1.6 | 1.12 | 1.3 |
| `SAVE` $x, Q^2$ (fast) | 1.5 | — | 1.3 | — |
| fastest `STRUCTM` | 1.65 | 1.7 | 1.53 | 1.4 |
| fastest times | 64 s | 50 s | 75 s | 55 s |

Table 6

Fraction of time spent evaluating PDF functions using an optimized `STRUCTM` for three programs: the next-to-leading-order jet calculation `ZTOP`, and two showering event generators, `HERWIG` and `PYTHIA`.

| Program | CTEQ4/5 | CTEQ6 |
| --- | --- | --- |
| `ZTOP` | 42% | 48% |
| `HERWIG` | 30% | 23% |
| `PYTHIA` | 9% | 9% |

## 4  Conclusions

Given the increasingly complex nature of calculations of hadronic physics we should investigate where bottlenecks in computational speed arise. It appears that one source of significant loss of computational speed is in evaluating parton distribution functions. For users of the CTEQ PDFs I propose three levels of optimization that can reduce computational times by up to a factor of 1.3–2.5 in showing event generators, and up to a factor of 4–5 in matrix-element calculations.

We observe in Sec. 3 that replacing `POLINT` with `POLINT3/4` greatly reduces the dependence of program execution time on the choice of compiler flags. This is a simple change, and can improve running times of some programs by up to a factor of 2. Since most programs call several PDFs in a row with the same values of $x$ and $Q^2$, the obvious next step is to modify the CTEQ routines `PARTONXN` to save previous results, and calculate only what has changed. The third level of optimization replaces the typical `CERNLIB` interface functions

10

`STRUCTM` and `PFTOPDG` [10] with fully optimized versions that are specialized to the CTEQ parton distributions. I recommend that these improvements be incorporated into the base CTEQ distribution [3], the `PDFLIB` routines in `CERNLIB` [10], and the new Les Houches Accord compilation of PDFs `LHAPDF` [13]. Full versions of the routines presented here may be obtained from the author, or from Ref. [11].

Despite these impressive gains, we see in Table 6 that evaluating PDFs remains the most time-consuming aspect of hadronic calculations. This suggests two avenues of investigation that should be considered for future programs. First, a systematic study of PDF evolution codes should be performed to determine whether there are more efficient interpolation algorithms to use with table-based PDFs. This would allow universal improvements in code execution. Second, each Monte Carlo writer should be aware of the timing issues (and potential bugs if $\bar{s}$ is not the same as $s$), and consider using an interface structure other than `STRUCTM`. The potential savings from calling one PDF instead of eight or more could be very significant. Finally, it is interesting that the optimizations I have listed can remove the apparent need for parameterizations of the parton distribution functions. In general, any calculation that relies heavily on interpolation, or multiple evaluations of a function in which some pieces do not vary, should see similar improvements in performance by applying these same techniques.

## Acknowledgements

## A   A fast `POLINT3`

This is a "fast" version of `POLINT3`, which has been optimized for the special case of 3-point fitting, and no possibility of divisions by zero. An error estimate is not returned, since it is never used in the CTEQ evolution codes. This code should be used to replace the version of `POLINT` called by the CTEQ4 and CTEQ5 PDFs.

```
C  This is a specialized recoding of Neville's
C   algorithm based on the POLINT routine from
C   "Numerical Recipes", but assuming N=3, and
C   ignoring the error estimation.
C  Written by Z. Sullivan, May 2004
```

```fortran
C  This file uses a minimal number of
C   instructions to do 3-point fitting.
C     SUBROUTINE POLINT  (XA,YA,3,X,Y,IGNORED)
      SUBROUTINE POLINT3 (XA,YA,N,X,Y,DY)
      IMPLICIT NONE
      DOUBLE PRECISION XA(3),YA(3),X,Y,DY,DEN
      DOUBLE PRECISION C1,HO,HP,HP2,W,D1,D2
      INTEGER N

      HO=XA(1)-X
      HP=XA(2)-X
      W=YA(2)-YA(1)
      DEN=HO-HP
      DEN=W/DEN
      D1=HP*DEN
      C1=HO*DEN

      HP2=XA(3)-X
      W=YA(3)-YA(2)
      DEN=HP-HP2
      DEN=W/DEN
      D2=HP2*DEN

      W=HP*DEN-D1
      DEN=HO-HP2

      IF((X+X-XA(2)-XA(3)).GT.0D0) THEN
         Y=YA(3)+D2+HP2*W/DEN
      ELSEIF((X+X-XA(1)-XA(2)).GT.0D0) THEN
         Y=YA(2)+D1+HO*W/DEN
      ELSE
         Y=YA(1)+C1+HO*W/DEN
      ENDIF

      RETURN
      END
```

## References

[1]  B. W. Harris, E. Laenen, L. Phaf, Z. Sullivan, and S. Weinzierl, Phys. Rev. D **66**, 054024 (2002).

[2]  Zack Sullivan, Phys. Rev. D **70**, 114012 (2004).

[3] CTEQ Collaboration, H. L. Lai *et al.*, Phys. Rev. D **55**, 1280 (1997); Z. Phys. C **74**, 463 (1997); Eur. Phys. J. C **12**, 375 (2000).

[4] CTEQ Collaboration, J. Pumplin *et al.*, J. High Energy Phys. **07**, 012 (2002).

[5] G. Corcella *et al.*, J. High Energy Phys. **01**, 010 (2001).

[6] T. Sjöstrand *et al.*, Comput. Phys. Commun. **135**, 238 (2001).

[7] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery, *Numerical Recipes in FORTRAN: The Art of Scientific Computing* (Cambridge University Press, Cambridge, 1986), Chap. 3.1.

[8] Zack Sullivan, *Faster parton distribution evaluation in Monte Carlos*, FERMILAB-FN-755, hep-ph/0403055.

[9] *Intel Architecture Optimization Reference Manual*, Intel Corporation Document 245127-001, 1999; *Intel Fortran Compiler for Linux Systems User's Guide, Volume II: Optimizing Applications*, Intel Corporation Document 2553260-002, 2003.

[10] H. Plothow-Besch, W5051 PDFLIB 1991.03.21, CERN-PPE; Comput. Phys. Commun. **75**, 396 (1993); Int. J. Mod. Phys. **A10**, 2901 (1995); W5051 PDFLIB 2000.04.17, CERN-PPE.

[11] All files are available directly from the author, or from http://home.fnal.gov/~zack/pdf/.

[12] J. S. Conway *et al.*, *Report of the Tevatron Higgs working group*, in *Physics at Run II: the Supersymmetry/Higgs Workshop*, Fermilab, 1998, edited by M. Carena and J. Lykken (Fermilab, Batavia, 2002), p. 39, hep-ph/0010338.

[13] W. Giele *et al.*, *The QCD/SM working group: Summary report*, in *Les Houches 2001: Physics at TeV colliders*, Les Houches, France, 2001, edited by P. Aurenche *et al.*, p. 275, hep-ph/0204316.